The Technion – Israel Institute of Technology
Electrical Engineering Faculty
Computer Networks Laboratory

Semester winter 2000/2001

Routing in Bluetooth Scatternet

Yonatan Maman

Eishay Smith

Supervisor

Gil Zussman

Index

Index 2

1 Introduction 5

1.1 Project Purpose 5

1.2 Bluetooth 6

1.2.1 Overview 6

1.2.1.1 Hardware 6

1.2.1.1.1 Radio layer 6

1.2.1.1.2 Baseband 6

1.2.1.2 Stack 7

1.2.2 Network 8

1.2.2.1 Piconet 8

1.2.2.2 Scatternet 8

2 Algorithm 9

2.1 The algorithm main steps: 9

2.2 Application Algorithm 9

2.2.1 Read data use case 9

2.2.2 Forward use case 10

2.2.3 Inquiry use case 10

2.2.4 Sending use case 11

2.2.5 Receiving Ack case 11

2.3 Routing Algorithm 12

2.3.1 Network radius 12

2.3.1.1 Definition: 12
2.3.2 Motivation 12
2.4 Routing loops 12
2.5 Example 13
3 Software 14
3.1 General 14
3.1.1 Tools 14
3.1.2 OS 14
3.1.3 Java 14
3.2 Overview 15
3.2.1 Axis Stack 15
3.2.2 JNI 16
3.2.3 Object Serialization 18
3.2.4 Swing (JFC) 18
3.3 Application 19
3.3.1 General 19
3.3.1.1 Problems we deal with 19
3.3.2 Design 21
3.3.2.1 Layer diagram 21
3.3.2.2 Native Side 23
3.3.2.3 Java Side 24
3.3.2.3.1 Java BT Stack 24
3.3.2.3.1.1 StackJavaInterface 25
3.3.2.3.1.2 Inquiry Demon 26
3.3.2.3.1.3 Read Demon 27

3.3.2.3.1.4 Send Demon 28
3.3.2.3.1.5 Envelope 29
3.3.2.3.2 Routing 30
3.3.2.3.2.1 Rout table 30
3.3.2.3.2.2 Updating the table 31
3.3.2.3.2.3 Sending an envelop to its destination 31
3.3.2.3.2.4 Routing an envelope 32
3.3.2.3.2.5 Table broadcast routine 32
3.3.2.3.3 GUI 33

4 User Manual 35

4.1 Installation 35

4.2 How to use 36

4.2.1 How to send text messages 36
4.2.2 How to send text messages and Pictures attached 37
4.2.3 The View Panels. 38
4.2.4 The Configuration Panel. 39
4.2.5 The Properties Panel. 40

5 Bibliography 41

1 Introduction

1.1 Project Purpose

The Bluetooth network is based on piconets. A collection of connected piconets forms a scatternet. The communication within a piconets is a solved problem integrated within the Bluetooth stack.

We see the Bluetooth world as a collection of piconets that communicates with each other. We don’t get lower from the piconet resolution. This kind of configuration raises problems of communication between piconets that are not in broadcast range from each other.

Another question that was raised is how two piconets can know about each other if they are not in the Bluetooth range. And when they know of each other, how does piconet X know where to send a message so it will eventually arrive to piconet Y.

We suggest an efficient solution to the above problems.

1.2 Bluetooth

1.2.1 Overview

Bluetooth wireless technology is a de facto standard, as well as a specification for small-form factor, low-cost, short-range radio links between mobile PCs, mobile phones and other portable devices.

1.2.1.1 Hardware

1.2.1.1.1 Radio layer

This link operates in the unlicensed ISM band around 2.4GHz and uses spread spectrum communication. The band extends from 2400 to 2483.5 MHz in a vast majority of countries and this whole range is utilized for optimizing the spectrum spreading. However, for some countries with a smaller ISM band a downscaled version is also provided. For spread spectrum, the frequency hopping (FH) technique is used. As multiple uncoordinated networks may exist in this band and cause interference, fast FH and short data packets are used. As the error rate may be high, especially due to strong interference from microwave ovens, which operate at this frequency. CVSD coding has been adopted for voice, which can withstand high bit error rates. Also the packet headers are protected by a highly redundant error correction scheme to make them robust to errors.

1.2.1.1.2 Baseband

The baseband is the layer that controls the radio. The frequency hop sequences are provided by this layer. Baseband also takes care of lower level encryption for secure links. The packet handling over the wireless link is the responsibility of Baseband. Two types of links can be established:

SCO: Synchronous Connection Oriented. These links are meant for synchronous data-typically voice.

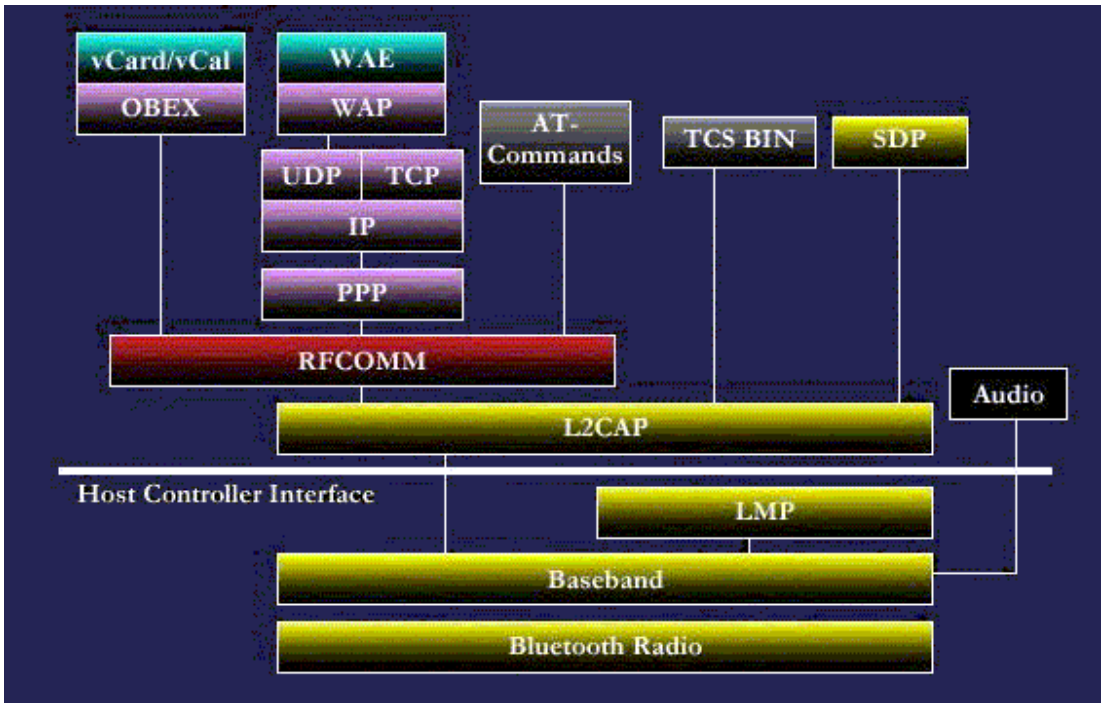
ACL: Asynchronous Connection less. These links may be used for data transfer applications, which do not require a synchronous link.

The baseband provides the functionalities required for devices to synchronize their clocks and establish connections. Inquiry procedures for discovering the addresses of devices in proximity are also provided. Error correction for packets is provided depending on the type of packet. Various packet types are specified for some common applications, differing in their data

capacity and error correction overheads. Five different channel types are provided for control information, link management information, user synchronous data, user asynchronous data and isosynchronous data. Data whitening is also carried out at this layer. The functions required for generating encryption keys and link keys are defined. A more detailed description of some of the baseband operations related to connection establishment is provided in section.

1.2.1.2 Stack

A general view on the stack:



We used mainly the Host Controller Interface (HCI) and the L2CAP. More information about the stack layer can be found at chapter 4.2.1.

1.2.2 Network

1.2.2.1 Piconet

A collection of devices connected via Bluetooth technology in an ad hoc fashion. A piconet starts with two connected devices, such as a portable PC and a mobile phone. The limit is set at 8 units in a piconet (that's why the required address-space is limited to 3 bits). All Bluetooth devices are peer units and have identical implementations. However, when establishing a piconet, one unit will act as a master for synchronization purposes, and the other unit(s) will be slave(s) for the duration of the piconet connection.

1.2.2.2 Scatternet

Multiple piconets may cover the same area. Since each piconet has a different master, the piconets hop independently, each with their own channel hopping sequence and phase as determined by the respective master. In addition, the packets carried on the channels are preceded by different channel access codes as determined by the master device addresses. As more piconets are added, the probability of collisions increases; a graceful degradation of performance results as is common in frequency hopping spread spectrum systems. If multiple piconets cover the same area, a unit can participate in two or more overlaying piconets by applying time multiplexing. To participate on the proper channel, it should use the associated master device address and proper clock offset to obtain the correct phase. A Bluetooth unit can act as a slave in several piconets, but only as a master in a single piconet. Since two piconets with the same master are synchronized and use the same hopping sequence, they are at the same piconet. A group of piconets in which connections consists between different piconets is called a **scatternet**.

A master or slave can become a slave in another piconet by being paged by the master of this other piconet. On the other hand, a unit participating in one piconet can page the master or slave of another piconet. Since the paging unit always starts out as master, a master-slave role exchange is required if a slave role is desired.

2 Algorithm

The routing algorithm is based on the known **Asynchronous Distributed Bellman Ford** algorithm. Our application is running on each master of a piconet. The communication within the piconet is not part of the algorithm.

2.1 The algorithm main steps:

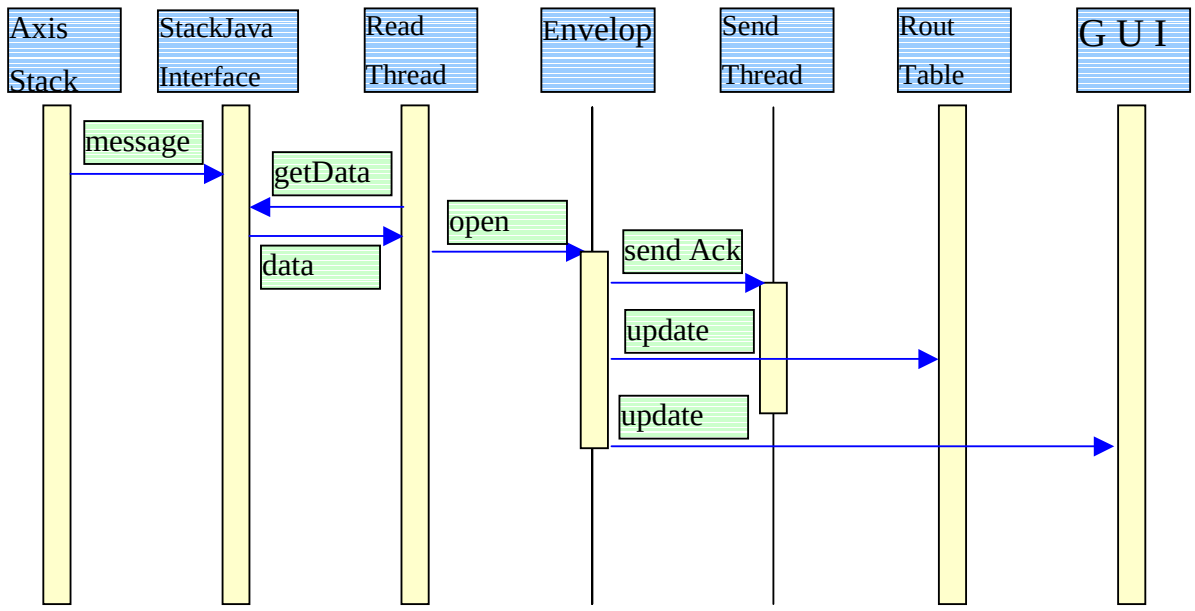
1. Every unit (Master in the bluetooth) is maintaining a routing table.
2. Once in a while the unit performs an inquiry and updates the table.
3. Every unit that has a distant greater then R (the network radius) is omitted from the table.
4. The algorithm looks for a routing loop and once one is found we solve it.
5. Any information exchange between two units in involved with mutual table update.
6. When a unit has a change in its table it spreads the update to its close neighbors.

2.2 Application Algorithm

We specify the Application Algorithm by describing three major use cases.

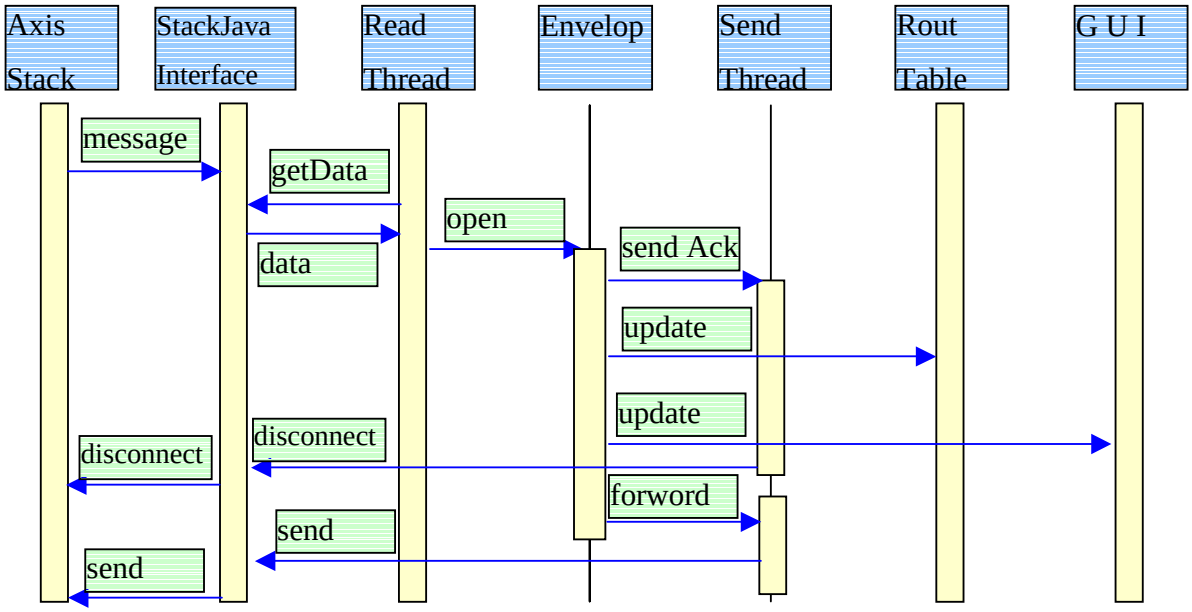
2.2.1 Read data use case

Description of the application flow chart, when data is arrived to the Bluetooth device.



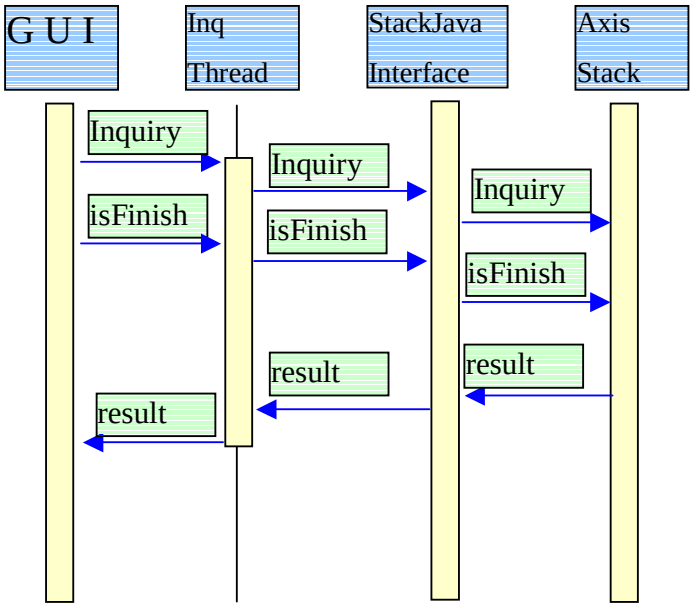
2.2.2 Forward use case

Description of the application flow chart, when a forward is performed.



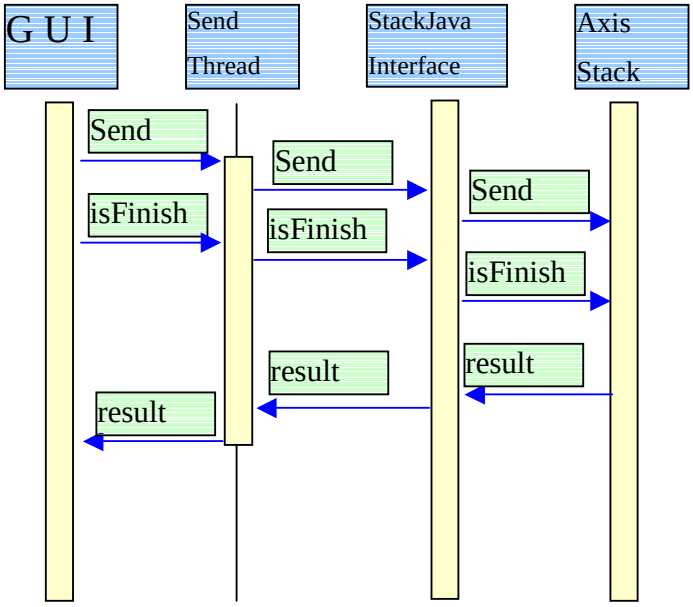
2.2.3 Inquiry use case

Description of the application flow chart, when Inquiry is performed.



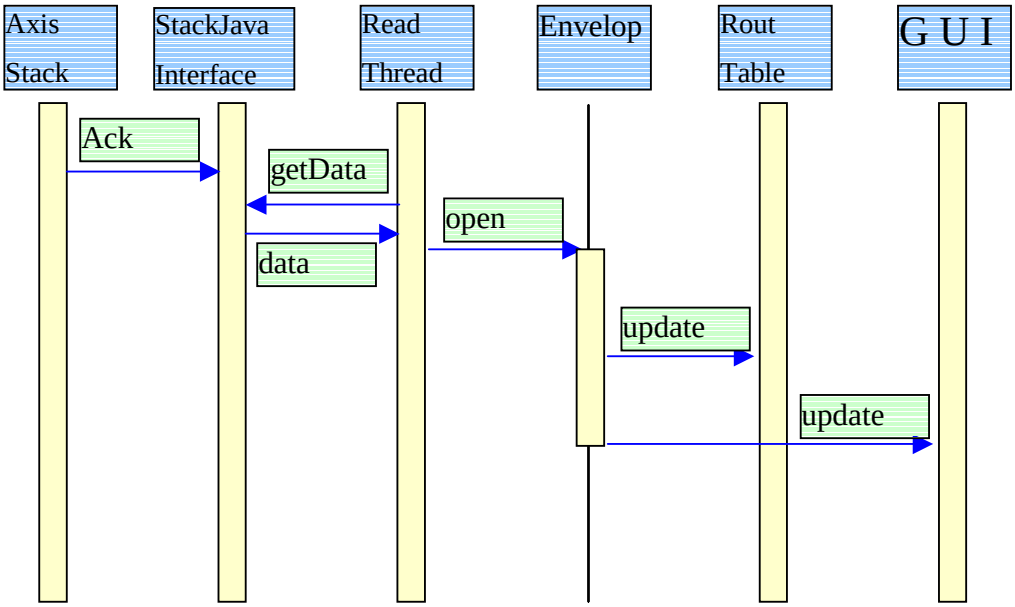
2.2.4 Sending use case

Description of the application flow chart, when Send is performed.



2.2.5 Receiving Ack case

Description of the application flow chart, when Ack is reached to the Bluetooth.



2.3 Routing Algorithm

2.3.1 Network radius

2.3.1.1 Definition:

Say piconet P has a "*net with radius R*" if between P and every other piconet in the network, the distance (in edges) is at most R.

2.3.2 Motivation

The network radius keeps the size of the network that a piconet knows about in bounds. Information about piconet that if out of the network radius range is probably irrelevant. This assumption is based of the dynamic behavior of the bluetooth network.

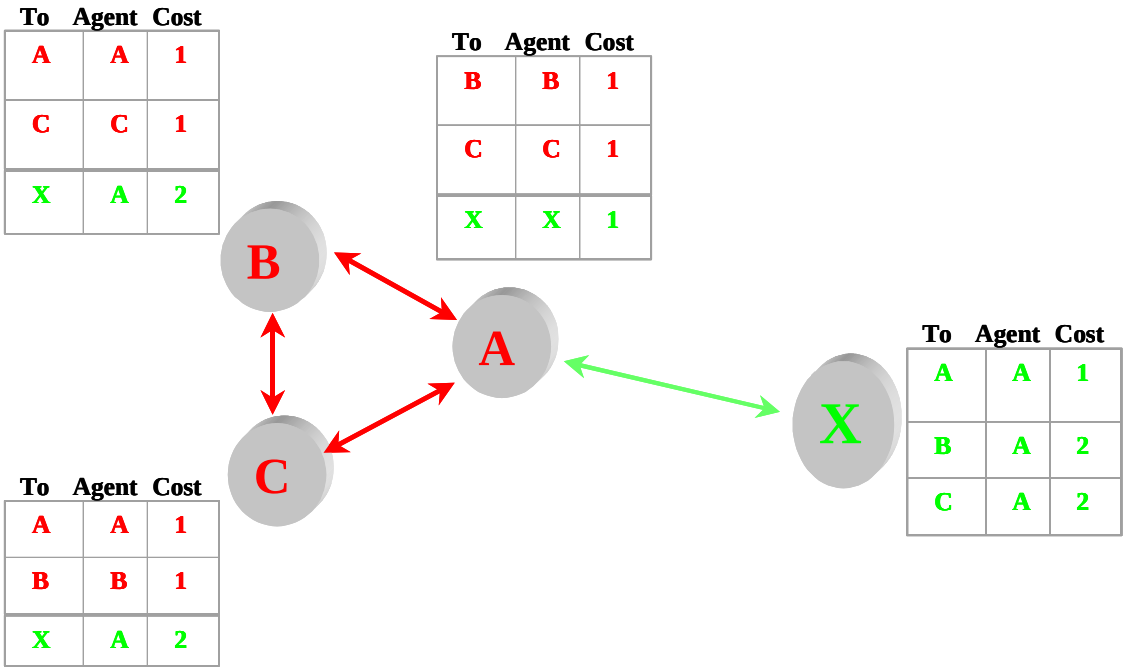
Lets consider the case where we don't have a network radius. There can b a theoretical situation that a piconet **A** knows of a piconet **B** that is 1000 piconets far away from it. At the time **A** got the information about **B**, **B** has changed it's location. This happened since the time that took the information to pass from **B** to **A** is very long.

2.4 Routing loops

Each time the table updates a simple check for routing loop is done. We count the number of edges that is needed to get to each piconet. If the number of edges is greater than the number of piconets that the table contains then there must be a routing loop. When such a loop is discovered then the problematic piconet is removed from the table.

2.5 Example

Consider the following situation. All the red markups are the initial stats before X arrives. You can see that piconets A, B and C are aware of each other and their routing tables are updated accordingly.



Now piconet X arrives to the neighborhood.

A notice X and updates it's table accordingly. After that, A sends it's updated table to all the piconets it knows of (B, C and X). The piconets updates their own tables accordantly.

3 Software

3.1 General

3.1.1 Tools

The tools we used are:

- Xemacs for the source editing and as the developing environment.

3.1.2 OS

We used Linux 6.2 as the OS platform.

3.1.3 Java

We used JDK 1.2.2

The main JDK packages that we used are:

- JNI (Java Native Interface).
- SWING (GUI package).
- Java object serialization (to transfer objects through the bluetooth).

The application is JDK 1.2 compatible and above.

3.2 Overview

3.2.1 Axis Stack

Axis has developed a Bluetooth driver for Linux that can be used both in our Linux port as well as on an ordinary Linux PC. The code is developed for a Linux 2.0.38 kernel, but it should work for 2.2 as well.

This stack is also used for axis's Bluetooth Access Point product which creates local hot spots of high-speed wireless connectivity. These areas provide a wireless communications link to local networks and the Internet for Bluetooth enabled mobile devices.

The code is still under development so we did not expected fully functional software - especially the integration of the stack within the kernel is fluctuating. The source code to the Bluetooth driver from Axis is available in their site. All code is released under the GPL (GNU General Public License).

Applications can use the stack in 2 major ways:

- The stack can be built into a Driver object. This object is attached to the virtual device /dev/ttyBT0. As any other dev it supplies I/O methods such open / read / write etc. in this way the stack is run in kernel mode and it is more difficult to make changes.
- The stack can be built into a normal object. This object can be linked to any application (like our for example). In this mode the stack is run in user mode. To simulate the read method the stack creates a new thread called "hci_receive_thread". This thread takes car of reading data from the serial device and put it in the button of the stack (hci level).

3.2.2 JNI

The Java Native Interface (JNI) is the native programming interface for Java that is part of the JDK. By writing programs using the JNI, you ensure that your code is completely portable across all platforms.

The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly. In addition, the *Invocation API* allows you to embed the Java Virtual Machine into your native applications.

Programmers use the JNI to write native methods to handle those situations when an application cannot be written entirely in the Java programming language. For example, you may need to use native methods and the JNI in the following situations:

- The standard Java class library may not support the platform-dependent features needed by your application.
- You may already have a library or application written in another programming language and you wish to make it accessible to Java applications.
- You may want to implement a small portion of time-critical code in a lower-level programming language, such as assembly, and then have your Java application call these functions.

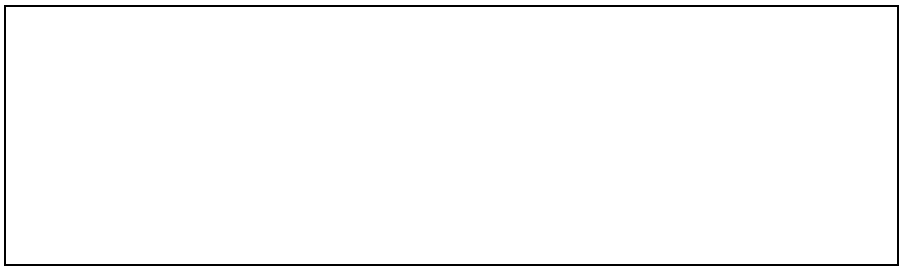
Programming through the JNI framework lets you use native methods to do many operations. Native methods may represent legacy applications or they may be written explicitly to solve a problem that is best handled outside of the Java programming environment.

The JNI framework lets your native method utilize Java objects in the same way that Java code uses these objects. A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code. A native method can even update Java objects that it created or that were passed to it, and these updated objects are available to the Java application. Thus, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

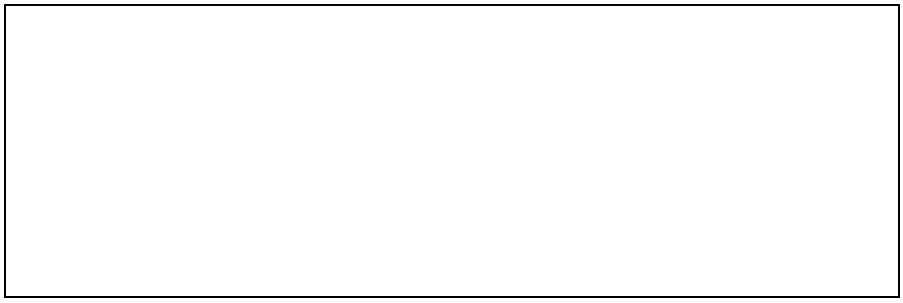
Native methods can also easily call Java methods. Often, you will already have developed a library of Java methods. Your native method does not need to "re-invent the wheel" to perform functionality already incorporated in existing Java methods. The native method, using the JNI framework, can call the existing Java method, pass it the required parameters, and get the results back when the method completes.

The JNI enables you to use the advantages of the Java programming language from your native method. In particular, you can catch and throw exceptions from the native method and have these exceptions handled in the Java application. Native methods can also get information about Java classes. By calling special JNI functions, native methods can load Java classes and obtain class information. Finally, native methods can use the JNI to perform runtime type checking.

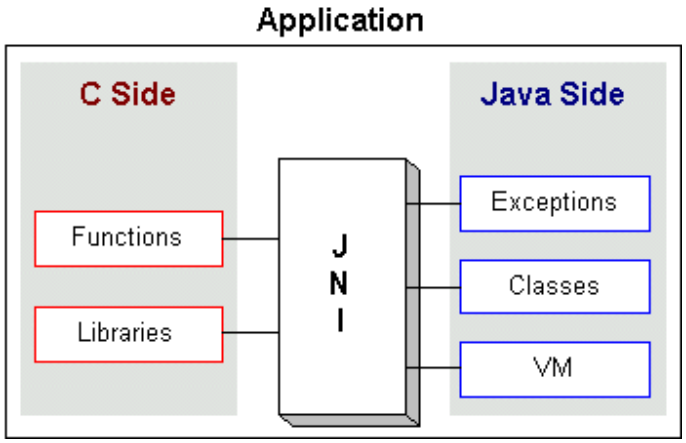
For example, the following figure shows how a legacy C program can use the JNI to link with Java libraries, call Java methods, use Java classes, and so on.



The next figure illustrates calling native language functions from a Java application. This diagram shows the many possibilities for utilizing the JNI from a Java program, including calling C routines, using C++ classes, calling assembler routines, and so on.



It is easy to see that the JNI serves as the glue between Java and native applications. The following diagram shows how the JNI ties the C side of an application to the Java side.



3.2.3 Object Serialization

The ability to store and retrieve Java objects is essential to building all but the most transient applications. The key to storing and retrieving objects in a serialized form is representing the state of objects sufficient to reconstruct the object(s). For Java objects, the serialized form must be able to identify and verify the Java class from which the contents of the object were saved and to restore the contents to a new instance. For Serializable objects, the stream includes sufficient information to restore the fields in the stream to a compatible version of the class.

Objects to be stored and retrieved frequently refer to other objects. Those other objects must be stored and retrieved at the same time to maintain the relationships between the objects. When an object is stored, all of the objects that are reachable from that object are stored as well.

3.2.4 Swing (JFC)

JFC is short for Java Foundation Classes, which encompass a group of features to help people build graphical user interfaces (GUIs). The JFC was first announced at the 1997 JavaOne developer conference.

3.3 Application

3.3.1 General

3.3.1.1 Problems we deal with

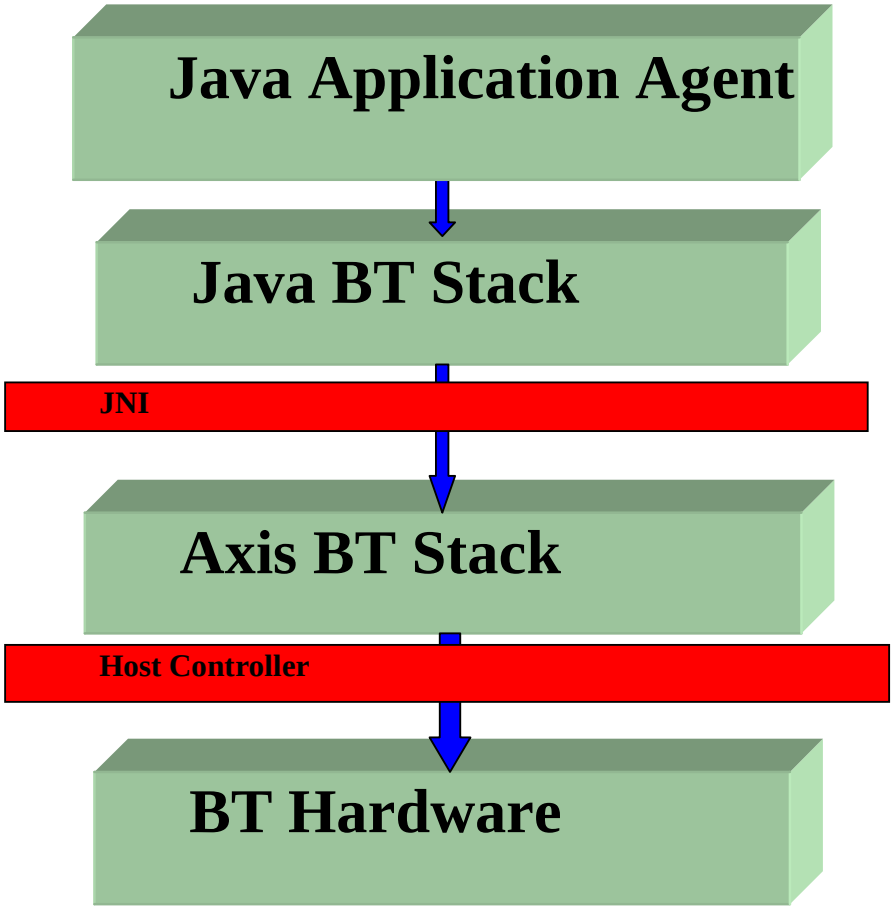
While working on this project we encountered with several problems.

- **Technical problems**:
  - At the first 4 weeks we had one digiAnswer unit and one Eriksson unit (for 2 days in a week). After a while we found that the digiAnswer unit is very unstable. In this setup we made the first connection and implemented a small chat utility.
  - The Bluetooth units cannot make more than one point-to-point communication. Therefore we break an existing connection before we attempt to make a new one.
- **Stack Problems:**
  - This stack implementation is not stable. Minor changes in the stack made some irreversible damage to the Linux kernel.
  - All the changes in the stack as described in chapter 4.3.2.2 were not trivial.
  - We had to find a way to distinguish between Bluetooth devices that run our application and all the other devices. We set a special class of device (C.O.D) for the Bluetooth device that attached to our application. When the application get the result of the inquiry (that include the Bluetooth address and its C.O.D) we filter out all the devices with a different C.O.D. we can say that the C.O.D is equivalent to port num in TCP/IP networks
  - The current stack has problems disconnecting from another unit. The only way we fount to solve this problem is to reset the stack when we wish to disconnect.

- Some of the hci commands don't perform "COMMAND_COMPLETE" event. We needed to add a mechanism that will notify the application when the command had performed.
- The mtu parameter (Minimal transfer Unit) is fixed to 127 bytes, and cannot be changed.
- The stack fragment messages larger than mtu but does not combine them back to the original message. We solved that in the application level.

- **Application problems:**
  - The major problem of writing this application was dealing with hybrid programming environment. We had to make a lot of research to integrate all the modules together: Bluetooth hardware, stack written in C, application written in Java, and all in Linux machine.

3.3.2 Design



3.3.2.1 Layer diagram

In the above diagram we describe the layer modulation of the system.

- The top layer is the **Java Application Agent** (JAA). This layer is written in java and holds the main logical part of the program. The application layer has all the tables and is in charge of the entire GUI. This layer deals is the only one that deals with the user interface.

- The **Java BT Stack** (JBTS) layer is a layer that acts like a Bluetooth driver for the Axis Stack. The JBTS communicates with the lower layers using the JNI technology.
- The **Axis BT Stack** (ABTS) layer is written in c. this layer is mostly the Bluetooth stack we got from Axis. Since this stack is open source we could change it so it will support the JNI calls. This way the BT stack can give the java stack Bluetooth services.
- The **Hardware** layer is the Bluetooth itself.

3.3.2.2 Native Side

We tried to keep the Stack Implementation as is and to create an interface so minimal changes need to be made in order to integrate with the stack. This way a new version of the stack can be easily replace the old one.

Most of the stack actions are asynchronous, i.e. when you ask for a service such as inquiry the result will not be returned right away. Instead of that the result would be prepared when the Bluetooth hardware will supply the answer. In some other cases the result of services were prompt on the output channel and not saved in any buffer.

We can divide most of our changes into 2 classes:

- Changing asynchronous methods to synchronous methods. We implemented that using a controlled polling mechanism.
- Adding buffers to hold information for the application. The result of inquiry service was manipulated this way.

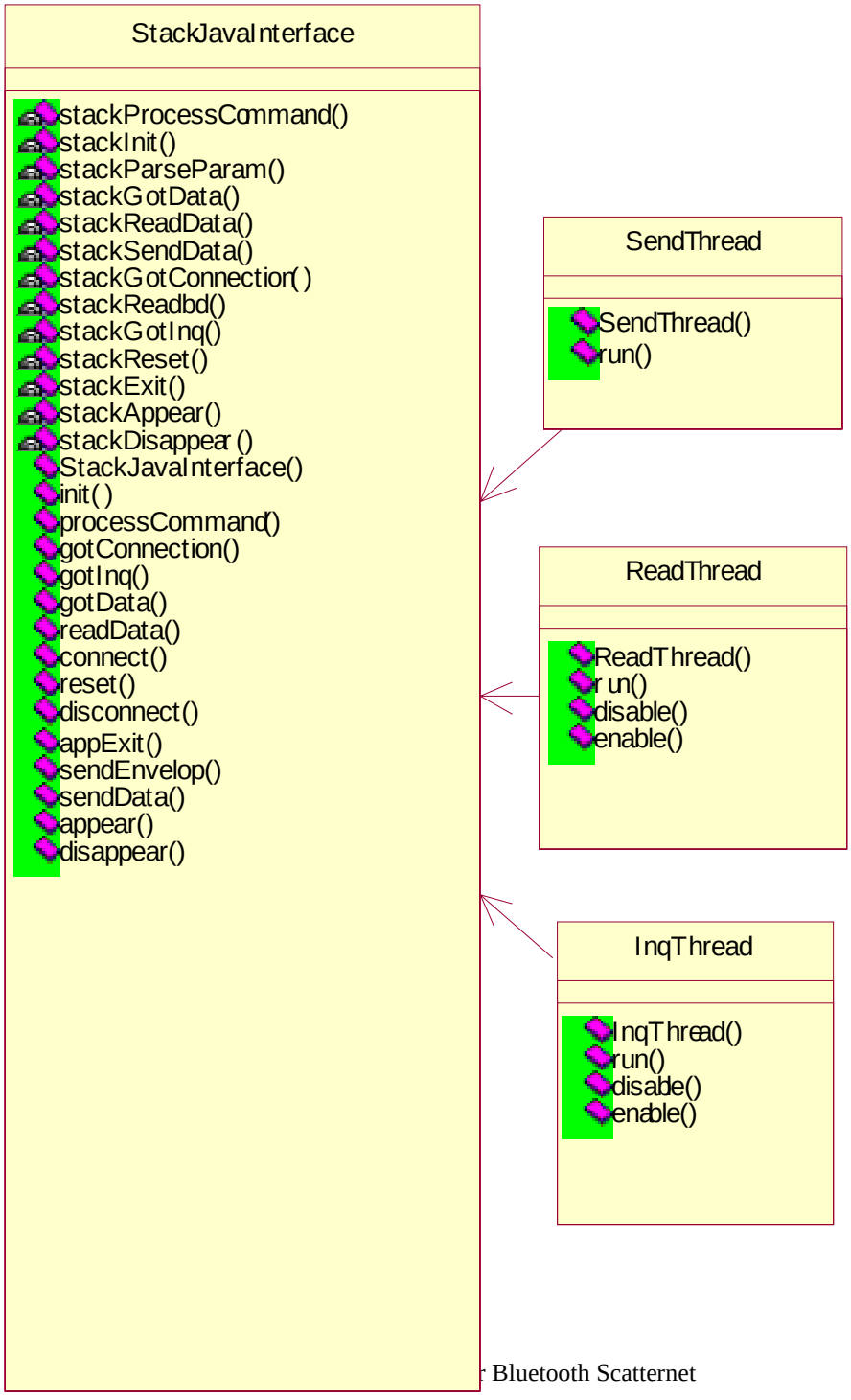
Here is a table that shows all the services used by the application (in fact it is the interface between the application and the stack), what kind of service: high level service or low level service, and what changes were made in the service.

Service	High / low level	Synchronous change	Storing data change
Stack init	High level	No	No
Stack reset	High level	No	No
Connect	High level	Yes	No
disconnect	High level	Yes	No
Receive data	High level	Yes	No
Send data	High level	Yes	Yes
Inquiry	High level (in fact hci level)	Yes	Yes
Read BT address	High level (in fact hci level)	No	Yes
Change class of device	Low level (hci)	No	No

3.3.2.3 Java Side

3.3.2.3.1 Java BT Stack

The communication between the routing logic and the Bluetooth stack is don via the Java BT Stack module. This module use the class StackJavaInterface and three demons (threads).



3.3.2.3.1.1 StackJavaInterface

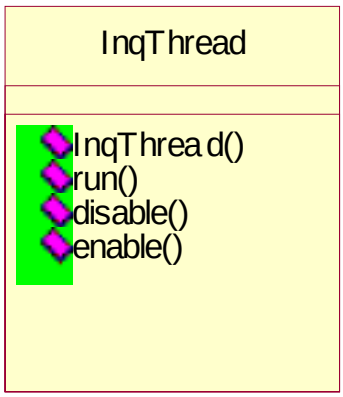
The StackJavaInterface class is the bootstrap of the driver. This is the class that does all the JNI work in the application. The class defines the following native methods to communicate with the Axis BT Stack:

```
int stackProcessCommand(String str);
int stackInit();
int stackParseParam();
int stackGotData();
int stackReadData(byte[] arr);
int stackSendData(byte[] arr, int len);
int stackGotConnection();
String stackReadbd();
String stackGotInq();
void stackReset();
void stackExit();
void stackAppear();
void stackDisappear();
```

The StackJavaInterface initiate the Stack and the JNI. After that it initiates the routing table with the local BTAddress of the Bluetooth hardware.

At the end, the java stack initiates and runs two of the demons – the Inquiry thread and the Read thread.

3.3.2.3.1.2 Inquiry Demon



The inquiry Demon is a thread of the class InqThread. This is a stand-alone thread. The thread does an inquiry every constant time interval. This interval can be configured through the GUI.

Each interval the demon goes to the StackJavaInterface object and asks it to invoke the processCommand method with the inquiry call. Then the demon goes to sleep for 15 seconds in which no more inquiry calls can be called. The sleeping time is for the stack to be able to response (we took safety overhead of three seconds). The inquiry call is a synchronic and therefore we can't get block on it and just get the answer back as the call returns.

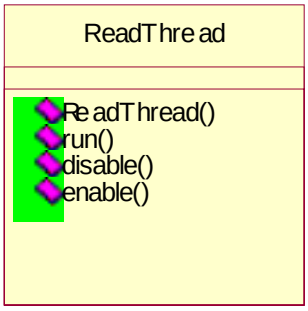
After the thread wake up from the 15 seconds of sleep, it checks for the results of the inquiry. It does that by calling the stackGotInq() method in the java stack. This call hoes to the Axis stack and checks for the inquiry result. If the return value is null, then the inquiry call has failed. Else, the call returns a value which is a string containing the details of the Bluetooth addresses that the inquiry has found.

When getting a result, the demon updates the routing table. The thread gets an indication whether the update has changed the table. If so, the demon will do a broadcast of the table to the neighboring piconets.

After finishing an inquiry call, the demon goes to sleep for a configurable time and when waking up, it does the inquiry again.

Note that an inquiry can be invoked manually through the GUI as well. Details on that see in the GUI section.

3.3.2.3.1.3 Read Demon



The read Demon is a thread of the class ReadThread. This demon is doing a poling task. Every configurable amount of time it does a pooling on the stack (using the JNI) and asks whether a new data has arrived to the stack. If no data has arrived, then the thread goes to sleep. When data arrives, the demon retrieves the data from the stack and makes a new Opener object with the data (in byte array form).

The data that we got from the stack is in fact a serialized java object of the Envelop class. First the opener retrieves the rout table from the envelope. Next the opener updates the GUI and the rout table with the new table it got from the sender.

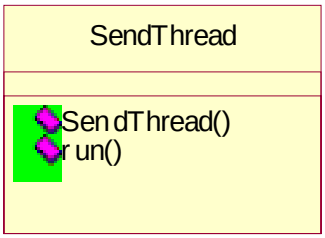
The received envelope can be one of two. It can either be simple message (text or a picture) or it can be an acknowledge (ack) message.

If the envelope is an ack message, then the opener updates the log and disconnect from the piconet that sent the message.

If the envelope is a simple message then the opener will be a little busier. First the opener will send an ack message back to the piconet that sent the message. The ack message will be a new envelope. To learn more about the envelope see the last chapter in the java stack. The opener uses a new send demon and submits the ack. Now the Opener checks if the table update had changed the table. If it did then a table broadcast routine is started (see in the routing chapter).

Now the opener makes a check if the envelope meant to be sent to this machine. The envelope destination may be another machine and this one is only a one of the routing stations in its way. In this case, the envelope will be sent now to its destination. If the envelope destination is this machine then the opener opens it and updates the GUI with the message and \ or the picture in it.

3.3.2.3.1.4 Send Demon



The send Demon is an object thread of the class SendThread. A new send thread is initialized and invoked for every new envelope to be sent.

If the envelope is an ack envelope then the communication to the receiver is still up. The demon will now simply send the envelope using the java stack.

If the envelope is not an ack envelop then there is no active Bluetooth connection to the receiver. The sender now will initiate a Bluetooth connection between the sender and the receiver using the java stack.

The connection initialization is an asynchronous call. The send thread cannot know when the connection will be established. Therefore the send thread do a pooling on the stack using the gotConnection() method in the java stack. This method goes to the BT stack and checks whether a connection has been made with the receiver. If a connection has not been maid after a timeout then the thread stops and declares an error.

When the connection to the receiver has been established, the sender just sends the envelope using the java stack. After sending the envelope the thread's roll is finished and it is finished.

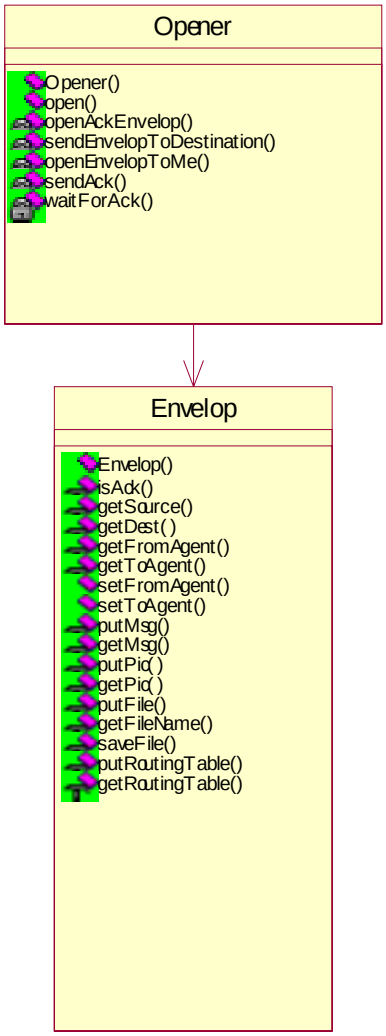
3.3.2.3.1.5 Envelope

The envelope class is a Serializable class that extends Hashtable.

An envelope contains the following information:

- RoutingTable – the current unit's table. We use any envelope to send the table to out neighbors and this way doing piggybacking.
- BTAddress of the source who sent the original message.
- BTAddress of the final destination of the original message.
- BTAddress of the current machine (fromAgent). Note that it is not necessarily the source of the message. It can be only a routing agent in the way.
- BTAddress of the machine that the message is sent to (toAgent). Note that it is not necessarily the destination of the message. It can be only a routing agent in the way.
- Boolean - true if the envelope is an acknowledge envelope.

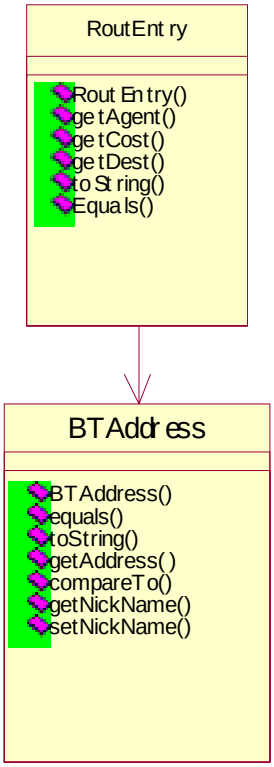
The envelope can contain a simple text message, a binary file, a java image object and the routing table.



3.3.2.3.2 Routing

3.3.2.3.2.1 Rout table

The rout table is the heart of the routing algorithm. The RoutTable holds a Hashtable that contains RoutEntry. Each RoutEntry holds the following members:



BTAddress dest –a Bluetooth machine address.

BTAddress agent – the Bluetooth address of the machine that is the agent of the dest machine.

int cost – the cost that it takes to get to the dest machine through the agent machine.

All the machines that the current unit knows about are have an entry in the Rout Table. For each of the machines the relevant entry is the one that it appears as the dest. The agent address is one that getting to dest using the agent takes the least cost. A private case is that the dest is also the agent. This means that there is no shorter way to get to the dest machine but to connect it directly. In another words, it is a close neighbor to the current machine.

The cost is the number of BT devices it gets to get to the specific dest machine. Cost = 1 if and only if the dest device is in Bluetooth range (about 10 meter) from the current device and the devices can communicate without any router.

3.3.2.3.2.2 Updating the table

Updating the table is the main routing bookkeeping of the routing module. Usually an update runs when the local unit got a new table from another close unit. We will call the other unit that sent the new table the “Sender”. Here we will describe the actions of the update method.

1. Checking to see if there are entries to remove from the table. The method goes through all of the entries in the table and checks whether there is an entry that its agent is the Sender. If we got one, we check if the Sender still knows the new unit we had found. If the Sender does not know about the dest of this entry then this entry must be deleted from the table.
2. Now we go through all of the entries in the new table.
    - If we find an entry that we don’t have in the table, we add it to the table indicating the Sender as the agent of this entry. The cost of this entry will be the cost it had in the old entry plus one.
    - If we find that the entry already exist in the table, we check its cost. If the cost through the Sender to the entry dest, we update the entry so the sender will be its agent and we also update the cost respectively.

    In addition, there is another rezone to call the update; this is done when we do an inquiry. The inquiry update reacts in a similar way to the last update we described.

3.3.2.3.2.3 Sending an envelop to its destination

When sending an envelope to a recipient. The sender goes through the routing table and looks for the relevant entry where the recipient is the dest. When the sender finds the entry it will put in the envelope to send the following addresses:

- BTAddress of the source who sent the original message is the current machine address.
- BTAddress of the final destination of the original message is the destination of the envelope.
- BTAddress of the current machine.
- BTAddress of the machine that the message is sent to (the agent). This is the agent as it appears in the dest entry in the rout table.

3.3.2.3.2.4 Routing an envelope

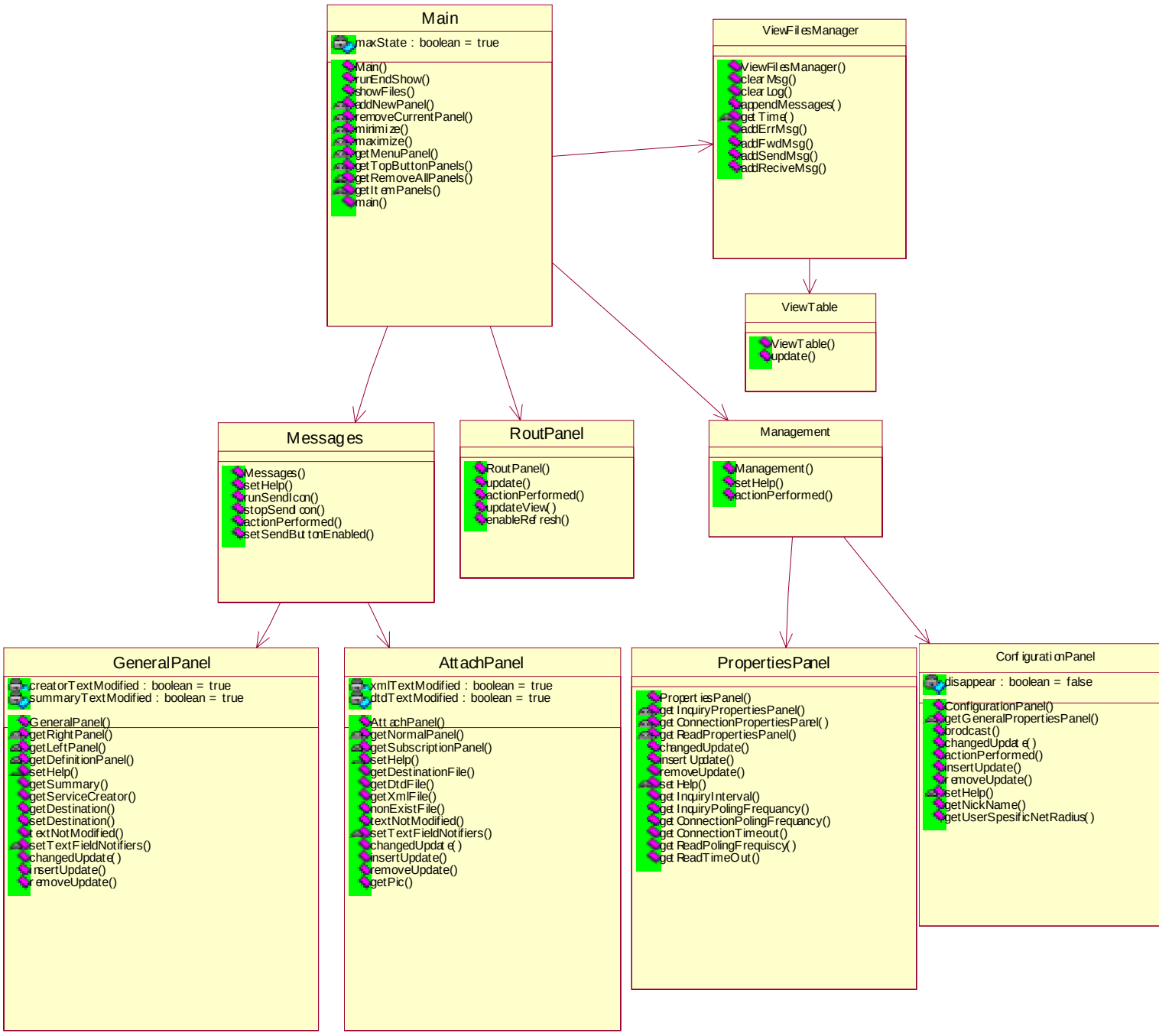
When routing an envelope from one Bluetooth unit to another. The router goes through the routing table and looks for the relevant entry where the “final destination” of the envelope is the dest. When the router finds the entry, it will put in the envelope to send the following addresses:

- BTAddress of the source who sent the original message will not change.
- BTAddress of the final destination of the original message will not change.
- BTAddress of the current machine is the router address.
- BTAddress of the machine that the message is sent to (the agent). This is the agent as it appears in the dest entry in the rout table.

3.3.2.3.2.5 Table broadcast routine

In every that the table change, the unit spreads the table to all of the close neighbors. One exception is made when the change comes from one of the close neighbors table update. In this case we will not send this one the table again (because it knows the update).

3.3.2.3.3 GUI



The GUI is built with the SWING package of the JDK. Each module of the in the diagram is a swing panel. The main panel is the Main class. This class holds all the other panels

as described in the class diagram. The main panels are the message panel that takes care of data to send. There you can compose the text and pick your attachment to send.

The configuration panel takes care of the information gathering for the java stack.

The view files manager takes care of the messages that the user will see and the view of the routing panel.

4 User Manual

4.1 Installation

Installation of this application is very simple. All you have to do is to have follow the following steps:

- Format your hard disk
- Install windows
- Format your hard disk
- Install linux
- download the file dynamicRouting.zip from: http://comnet.technion.ac.il/~cn4w01
- Unzip the file to your /root directory.
- Go to /root/proj/app directory and run “make”.
- The application is installed.

To use the application you need:

- Administrator permission on the linux machine.
- Jdk 1.2 (and up) installed.
- Bluetooth device connected to the computer.

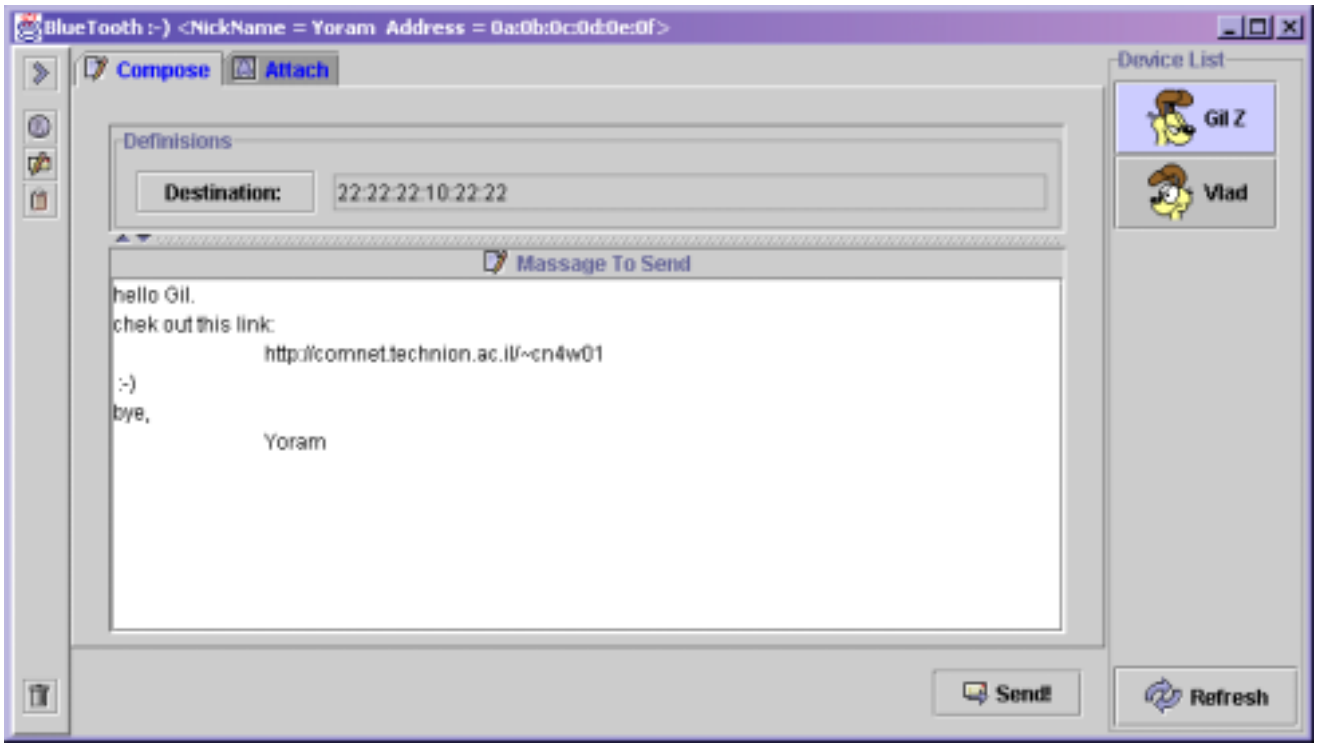
4.2 How to use

To run the application go to /root/proj/app directory and type “java Main”.

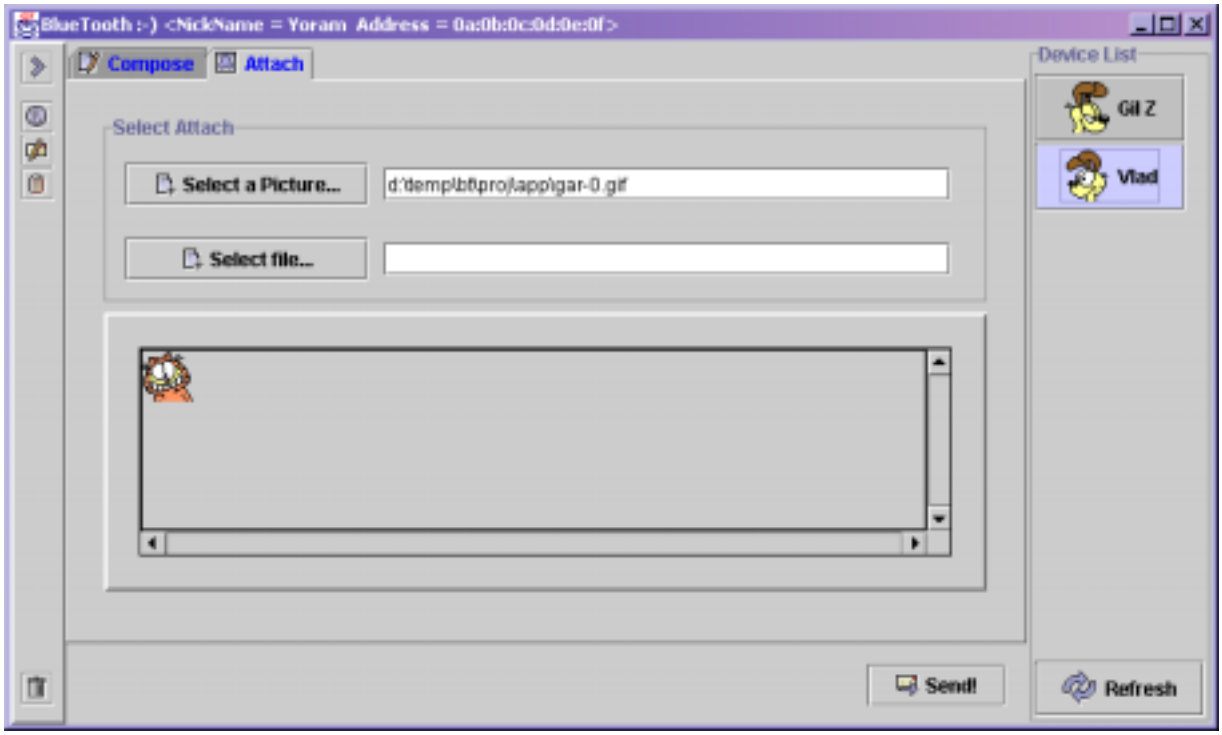
The application is based on graphic user interface. Here we will explain how to use it.

4.2.1 How to send text messages

Do inquiry by pressing the “Refresh” button. All the devices that your device recognizes will be shown in the device list. Choose one device, type your message and press the “Send” Button.



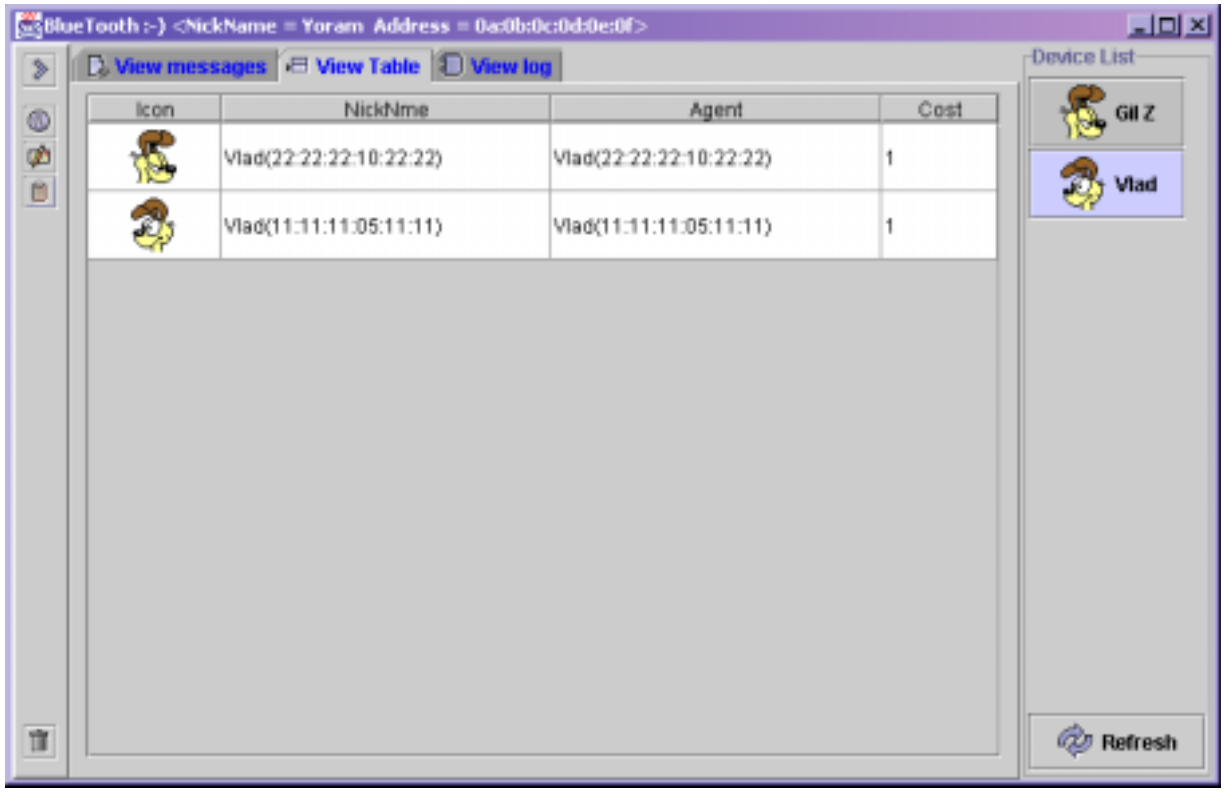
4.2.2 How to send text messages and Pictures attached



The same as text. Choose the “Attach” panel. Press the “Select Picture” Button and choose a picture file. Press “Send” to send.

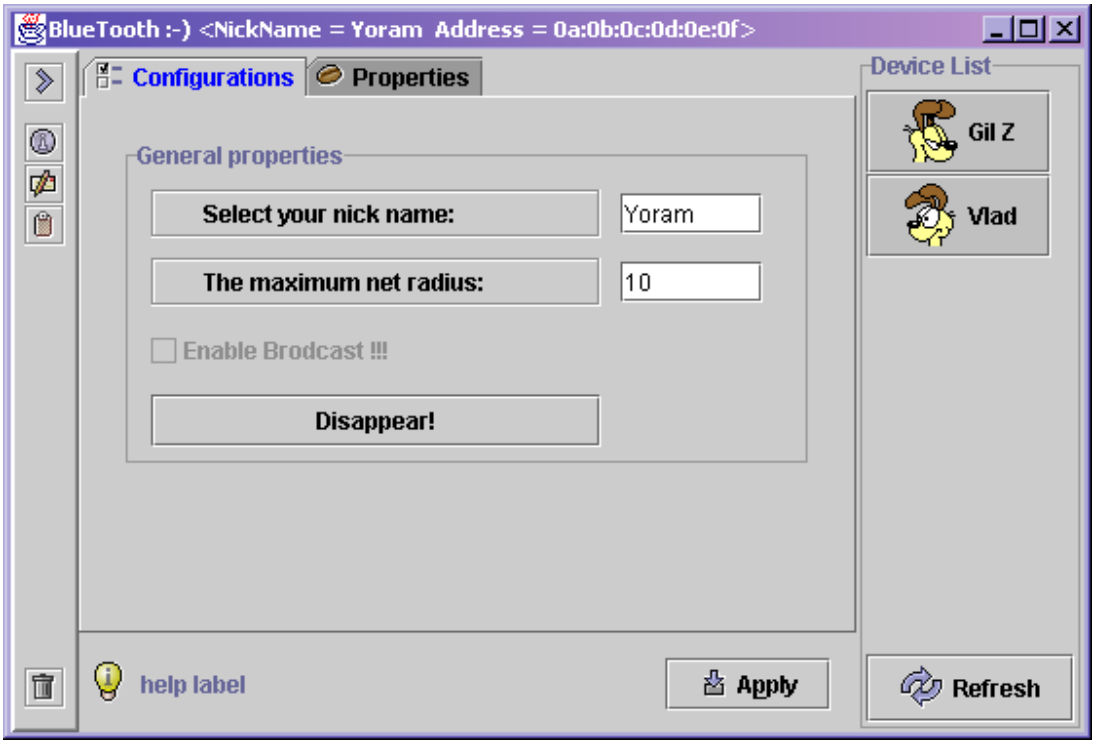
4.2.3 The View Panels.

Press the “View” button. You can choose one of 3 tabs: Messages, Table and Log.

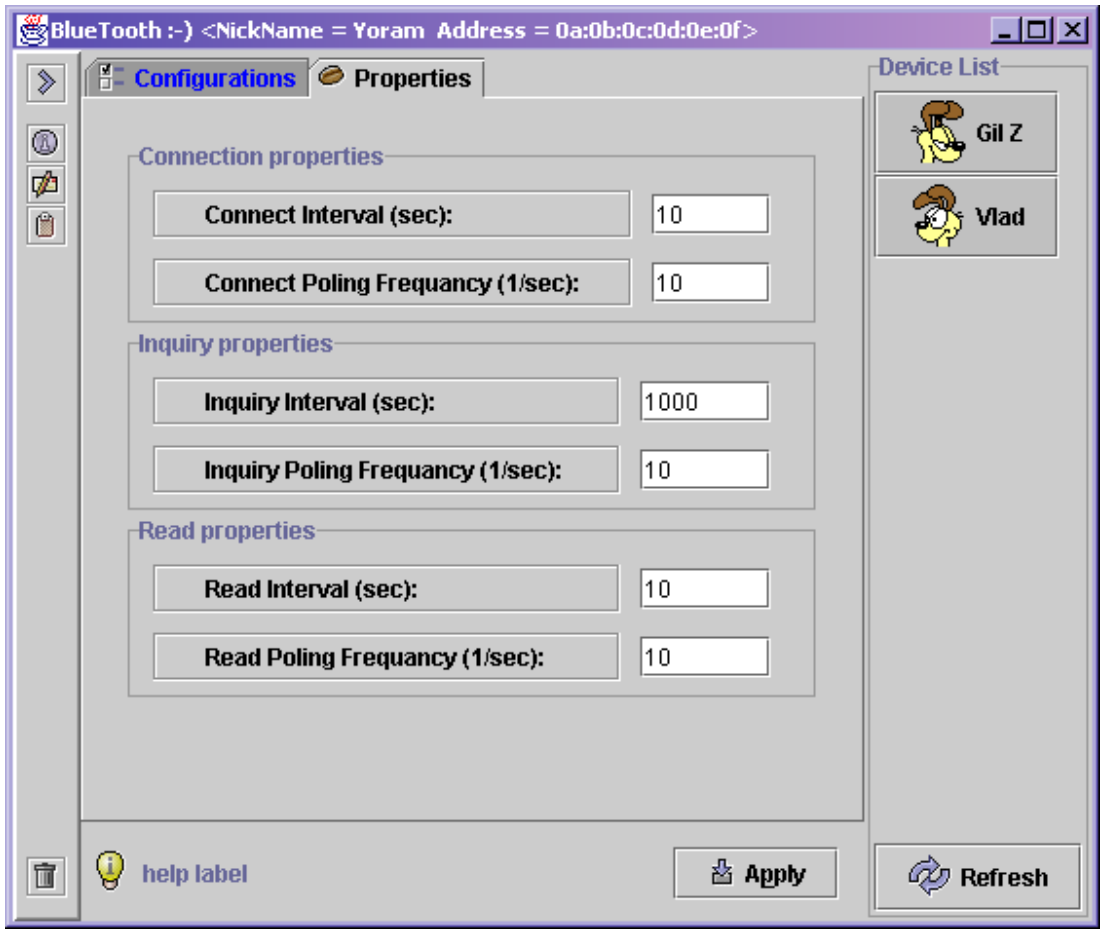


4.2.4 The Configuration Panel.

Press the “Options” button. You can choose one of 2 tabs: Configuration, Properties.



4.2.5 The Properties Panel.



5 Bibliography

http://www.bluetooth.com

http://www.motorola.com/bluetooth

http://www.zdnet.co.uk/news/specials/1999/04/bluetooth/

http://developer.axis.com/software/bluetooth/

http://www.linuxdoc.org